

Amendments to the Specification

Please replace paragraph [0003] with the following amended paragraph.

[0003] Managed runtime environments are typically implemented using a dynamic programming language such as, for example, ~~Java~~ JAVA programming language, C#, etc. A software engine (e.g., a Java Virtual Machine (JVM®) software engine, a Common Language Runtime (CLR) software engine, etc.), which is commonly referred to as a runtime environment, executes the dynamic program language instructions. The runtime environment interposes or interfaces between dynamic program language instructions (e.g., a ~~Java~~ JAVA program or source code) to be executed and the target execution platform (i.e., the hardware and operating system(s) of the computer executing the dynamic program) so that the dynamic program can be executed in a platform independent manner.

Please replace paragraph [0004] with the following amended paragraph.

[0004] Dynamic program language instructions (e.g., ~~Java~~-JAVA instructions) are not statically compiled and linked directly into native or machine code for execution by the target platform (i.e., the operating system and hardware of the target processing system or platform). Instead, dynamic program language instructions are statically compiled into an intermediate language (e.g., bytecodes) and the intermediate language may be interpreted or subsequently compiled by a just-in-time (JIT) compiler into native code or machine code that can be executed by the target processing system or platform. Typically, the JIT compiler is provided by a runtime environment that is hosted by the operating system of a target processing platform such as, for example, a computer system. Thus, the runtime environment and, in particular, the JIT compiler, translates platform independent program instructions (e.g., ~~Java~~-JAVA bytecodes, C# bytecodes, etc.) into native code (i.e., machine code that can be executed by an underlying target processing system or platform).

Please replace paragraph [0006] with the following amended paragraph.

[0006] Unfortunately, the processing overhead associated with object synchronization results in a significant reduction in execution time. For example, in the case of some well-known ~~Java~~-JAVA applications and benchmarks, synchronization overhead may consume between about ten to twenty percent of overall execution time. Furthermore, synchronization is usually employed as a safeguard to prevent contentions during runtime (particularly in the case of object libraries), regardless of whether such synchronization is actually required during runtime.

Please replace paragraph [0016] with the following amended paragraph.

[0016] FIG. 1 is a block diagram of an example architecture 100 that may be used to implement the generational escape analysis apparatus and methods described herein. For the example architecture 100, one or more software applications 102, which are composed of one or more dynamic programming languages and/or instructions, are provided to a language compiler 104. The applications 102 may be written using a platform independent language such as, for example, ~~Java~~ JAVA programming language or C#. However, any other dynamic or platform independent computer language or instructions could be used instead. In addition, some or all of the applications 102 may be stored within the system on which the applications are to be executed. Additionally or alternatively, some or all of the applications may be stored on a system that is separate (and possibly remotely located) from the system on which the applications 102 are to be executed.

Please replace paragraph [0017] with the following amended paragraph.

[0017] The language compiler 104 statically compiles one or more of the applications 102 to generate compiled code 106. The compiled code 106 is intermediate language code or instructions (e.g., bytecodes in the case where the complied application(s) are written in ~~Java~~ JAVA programming language) that is stored in a binary format in a memory (not shown). As with the applications 102, the compiled code 106 may be stored locally on a target system 108, on which the compiled code 106 is to be executed. The target system 108 may be a computer system or the like such as that described in greater detail below in connection with Fig. 7. The target system 108 may be associated with one or more end-users or the like. Additionally or alternatively, the compiled code 106 may be delivered to the target system 108 via a communication link or links including, for example, a local area network, the Internet, a cellular or other wireless communication system, etc.

Please replace paragraph [0018] with the following amended paragraph.

[0018] One or more portions of the compiled code 106 (e.g., one or more software applications) may be executed by the target system 108. In particular, an operating system 110 such as, for example, Windows, Linux, etc., hosts a runtime environment 112 that executes one or more portions of the compiled code 106. For example, in the case where the compiled code 106 includes ~~Java~~-JAVA bytecodes, the runtime environment 112 is based on a ~~Java~~-JAVA Virtual Machine (JVM®) software engine or the like that executes ~~Java~~-JAVA bytecodes. The runtime environment 112 loads one or more portions of the compiled code 106 (i.e., the intermediate language instructions or code) into a memory (not shown) accessible by the runtime environment 112. Preferably, the runtime environment 112 loads an entire application (or possibly multiple applications) into the memory and verifies the compiled or intermediate language code 106 for type safety.

Please replace paragraph [0020] with the following amended paragraph.

[0020] In general, dynamic programming languages such as, for example, ~~Java~~ JAVA programming language, provide synchronization features that enable software designers to generate thread-safe code or software objects. A synchronized software object can only be accessed by one execution thread at a time, thereby preventing a conflict or contention associated with arguments or variables used by the object from occurring. In other words, global objects and other objects accessible by more than one execution thread can be made thread safe by introducing software lock and unlock mechanisms that prevent more than one thread from simultaneously accessing the objects. However, use of synchronization imposes a significant overhead on applications.

Please replace paragraph [0021] with the following amended paragraph.

[0021] Escape analysis is a technique that eliminates unnecessary synchronization operations. In particular, an object may escape the method that created the object (i.e., the object is not local to the method). Alternatively, the object may escape the thread that created the object (i.e., other threads may access the object). Further, escape analysis may also guide allocation of stack objects to create ~~Java~~ JAVA objects on the stack rather than on the heap (i.e., an area of the main memory that a program may use to store data in a varying amount known only when the program is running).